**SupaBase DB Local Project Setup Prompt and steps:**

**A) One‑time prerequisites (local)**

1. Install Docker Desktop on your Mac and ensure it runs.
2. Install the Supabase CLI on your machine.
3. Confirm Node.js is installed (any current LTS is fine).

**B) Initialize a local Supabase workspace**

1. In your project folder, initialize a Supabase workspace (creates a supabase directory).
2. Start the local Supabase stack (Postgres, Auth, Storage, Realtime, Studio).
3. Open the local Studio URL (shown by the CLI) to verify services are running.

**C) Link the CLI to your existing cloud project (supabase.com)**

1. Log in to Supabase from the CLI using your Supabase account token.
2. Link your local workspace to the correct Supabase **Project Reference** (find it in your project’s settings on supabase.com).

**D) Adopt a “migration‑first” workflow**

1. Ensure your repository contains:
   * A supabase/migrations folder for timestamped SQL migrations.
   * A supabase/seed.sql file for **local/dev** sample data only.
   * A supabase/functions folder for Edge Functions (optional).
2. If your remote project already has schema but no local migrations:
   * Create a **baseline migration** from the remote schema.
   * Rebuild your local database from migrations to validate the baseline.

**E) Local development loop**

1. Create a new migration for any schema/policy/index change.
2. Apply migrations to your local DB.
3. (Optional) Load local seed data for testing.
4. Inspect and test using local Studio (tables, RLS, users, storage).
5. If using Edge Functions, serve and test them locally.
6. (Optional) Generate TypeScript types from the local DB for type‑safe development.
7. Commit your migration files, seeds (dev only), and any function changes.

**F) Prepare the GitHub repository**

1. Create (or connect) a GitHub repo for your project.
2. Push your code, including the supabase directory structure described above.
3. Protect the main branch (require pull requests and at least one review).
4. Define code ownership for the supabase/migrations path to enforce reviews (optional but recommended).

**G) Store secrets in GitHub (for CI/CD)**

1. In your GitHub repo settings, create **Actions Secrets**:
   * SUPABASE\_ACCESS\_TOKEN (create in supabase.com under “Access Tokens”).
   * SUPABASE\_PROJECT\_REF (from your Supabase project settings).
2. (Optional) If you want a staging environment, also add:
   * SUPABASE\_PROJECT\_REF\_STAGING.

**H) Continuous Integration (CI) on Pull Requests**

1. Add a workflow that:
   * Checks out your repo.
   * Installs the Supabase CLI.
   * Starts a temporary local Supabase stack.
   * Rebuilds a clean DB from your migrations and seed to ensure the SQL is valid.
   * (Optional) Runs your automated tests.
   * Stops the local stack.
2. Require this workflow to pass before merging PRs into main.

**I) Continuous Deployment (CD) on merge to**

**main**

1. Add a workflow that:
   * Checks out your repo on merges to main.
   * Installs the Supabase CLI.
   * Links to your **production** project using the project ref.
   * Pushes all pending DB migrations to the linked project.
   * Deploys all (or specific) Edge Functions if you use them.
   * (Optional) Sets or updates any runtime secrets needed by your functions.
2. Confirm the workflow uses only GitHub Secrets (no secrets in the repo).

**J) Optional: Staging environment**

1. Create a separate Supabase project for staging.
2. Add a second deploy workflow that triggers on a develop branch (or PR label) and targets the **staging** project ref.
3. Point your Preview app deploys (e.g., Vercel) to the staging project’s URL and anon key.

**K) App environment configuration (FYI, even if UI is separate)**

1. Local app uses local Supabase URL and anon key.
2. Preview app uses staging Supabase URL and anon key.
3. Production app uses production Supabase URL and anon key.
4. Keep the **service role key** server‑side only (never expose it to the browser).

**L) Backup & rollback**

1. Schedule periodic database dumps from the remote project.
2. For small corrections, create a new “fix” migration that reverses or adjusts the previous change.
3. For emergencies, restore from backups (align with your team’s RPO/RTO expectations).

**M) Guardrails and best practices**

1. Never edit production schema directly in the UI; always use migrations.
2. Keep seed.sql for local/dev only; production data changes belong in explicit migrations.
3. Review RLS policies like code—treat them as part of PRs.
4. Keep a concise README describing:
   * How to run local Supabase.
   * How to add a migration.
   * How to run CI locally if needed.
   * Where the GitHub workflows live and what they do.

**N) Day‑to‑day developer flow (summary)**

1. Create a feature branch.
2. Add a migration for any schema/policy change.
3. Apply locally, test in Studio, adjust as needed.
4. Commit and push; open a PR.
5. CI validates migrations on the PR.
6. Merge to main after review.
7. CD deploys migrations and functions to supabase.com automatically.
8. Verify changes in the Supabase dashboard (prod project).